



2183
10/046,200

1/7

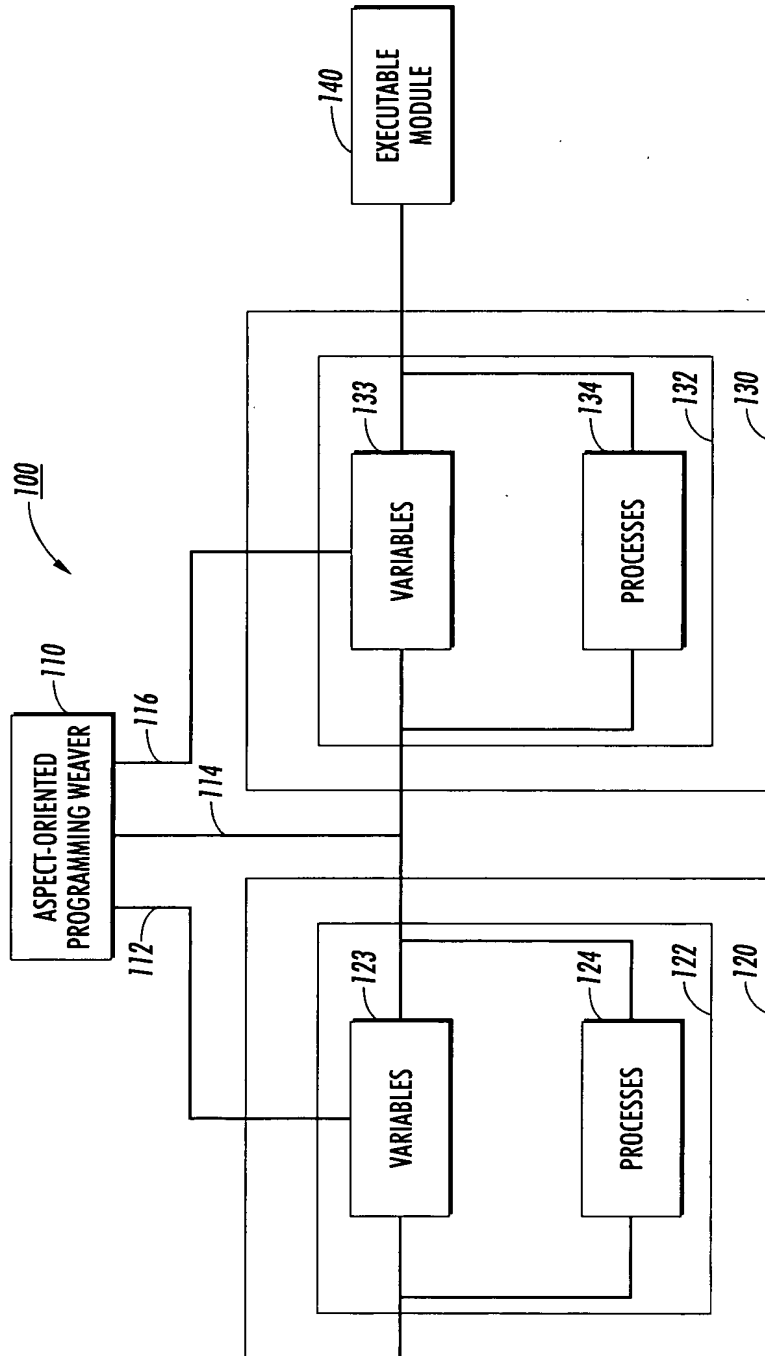


FIG. 1



2183
10/046,200

2/7

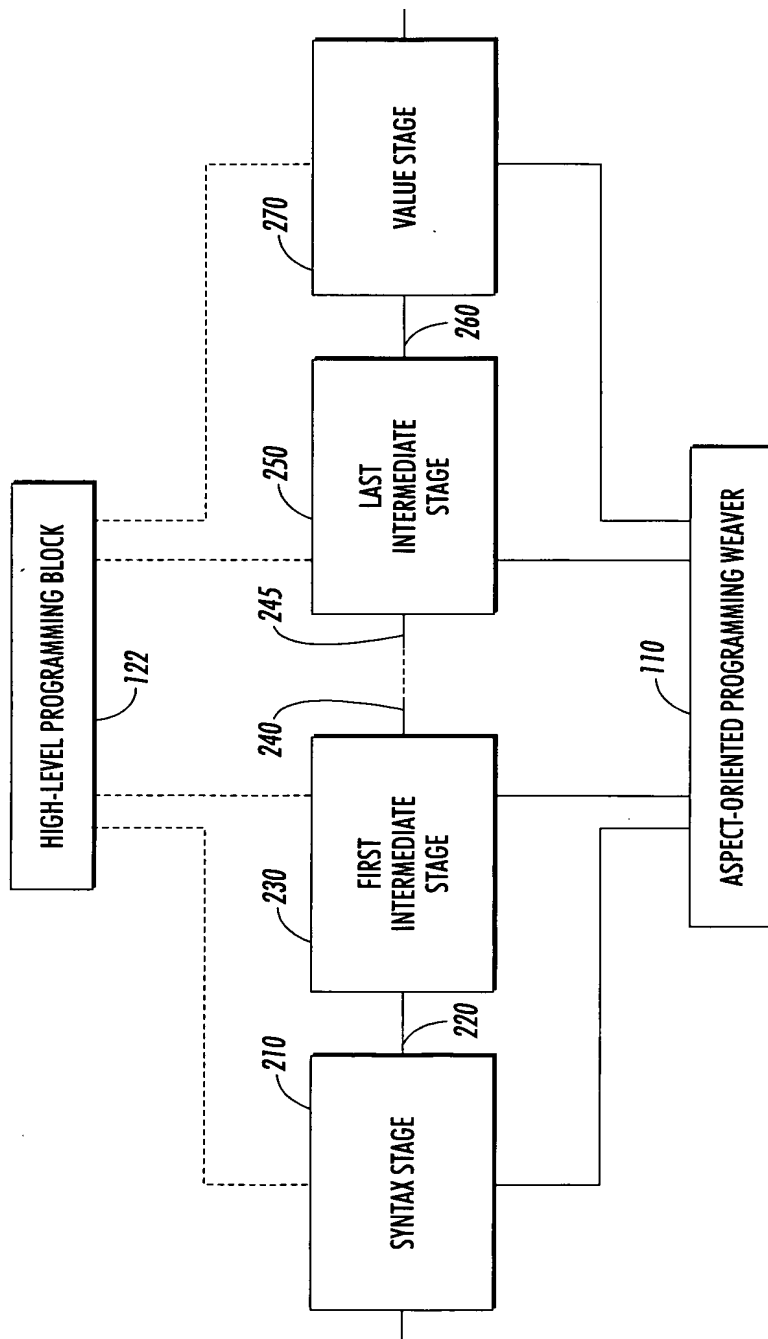


FIG. 2



```
1      (stage requested-loops)
2      (projection key :defined-on requested-loops)
3      (projection computing-loop :defined-on requested-loops)
4      (define test
5      (lambda (x y z)
6        (and! (and! X y) z)))
7      (define and!
8      (reduction-stage requested-loops
9      (lambda (arg1 arg2)
10       (pointwise #'and arg1 arg2))))
11     (propagator requested-loops :bottom-up
12     (lambda (term)
13       (case requested-loops term
14         ((pointwise op arg1 arg2) (op arg1 arg2)
15         (let ((starting-loop
16               (fuse-loops (get-or-make-loop arg1) (get-or-make-loop arg2))
17               (my-key (gensym)))
18           (deconstruct requested-loops starting-loop
19             (ptw-loop fn inputs outputs) (fn inputs outputs)
20             (let* ((new-fn (reduction-stage computation
21                           (lambda (args)
22                             (let* ((temp (fn args))
23                                   (result
24                                     (op (find (key arg1) temp)
25                                           (find (key arg2) temp))))
26                                   (cons (cons my-key result)
27                                         temp))))
28               (new-loop (defer (ptw-loop new-fn inputs outputs))))
29             (update (key value) my-key)
30             (update (computing-loop value) new-loop))
31             (if (computing-loop arg1)
32                 (update (computing-loop arg1)
33                           (defer (loop-reference value))))
34             (if (computing-loop arg2)
35                 (update (computing-loop arg2)
36                           (defer (loop-reference value))))))
37             (else (note-demands value)
38             )))
39
```

FIG. 3A



```
39      (define get-or-make-loop (value)
40      (if (and (same-frequency value) (computing-loop value))
41          (get-loop value)
42          (defer (ptw-loop
43                  (reduction-stage computation
44                    (lambda (args) args)
45                    (list (cons (key value) value))
46                      nil))))))

47      (define get-loop
48      (reduction-stage computation
49        (lambda (value)
50          (computing-loop (get-loop-location value))))))
51      (define get-loop-location
52      (reduction-stage computation
53        (lambda (value)
54          (case requested-loops (computing-loop value)
55            ((loop-reference next) (next)
56             (get-loop-location next))
57            (else value))))))
58      (define note-demands (value)
59      (case requested-loops value
60        ((fn . args) (fn args)
61                     (record-demand fn)
62                     (map args #record-demand)))
63        ((case stage value (pattern vars body) (else otherwise))
64         (stage value pattern vars body otherwise)
65         (record-demand value)
66         (record-demand body)
67         (record-demand otherwise))
68        ((lambda vars body) (vars body)
69         (record-demand body))))
70      (define record-demand (value)
71      (if (computing-loop value)
72          (let ((place (get-loop-location value))
73                (key (key value)))
74            (case requested-loops (computing-loop place)
75              ((ptw fn inputs outputs) (fn inputs outputs)
76               (if (not (member key outputs))
77                   (let ((new-outputs (cons key outputs)))
78                     (update (computing-loop place)
79                           (delay (ptw fn inputs new-outputs))))))))))
80
```

FIG. 3B



5/7

```
80 (define ptw-loop
81   (lambda (fn inputs outputs)
82     (let ((output-pairs (early-mapcar (reduction-stage computation
83                                       (lambda (key) (cons key (new-array)))
84                                       outputs))))
85       (dotimes ((i 0 99))
86         (let* ((input-scalars
87                 (early-mapcar (reduction-stage computation
88                               (lambda (pair)
89                                 (let ((key (first pair))
90                                     (array (second pair)))
91                                   (cons key (elt array i)
92                                     inputs))
93                               (output-scalars (fn input-scalars)))
94                 (early-map (reduction-stage computation
95                             (lambda (pair)
96                               (let ((key (first pair))
97                                   (array (second pair)))
98                                 (setf (elt array i)
99                                   (find key output-scalars))))
100                  output-pairs))))))
101 (define pointwise (fn op1 op2 > result)
102   (reduction-stage computation ;; inlineable after loop fusion
103     (find (key result) (get-loop result))))
104 (define fuse-loops
105   (lambda (loop1 loop2)
106     (if (stage-eq requested-loops loop1 loop2)
107         loop1
108         (deconstruct loop-structure loop1
109           ((ptw-loop fn1 inputss1 outputss1) (fn1 inputs1 outputs1)
110             (deconstruct loop-structure loop2
111               ((ptw-loop fn2 inputs2 outputs2) (fn2 inputs2 outputs2)
112                 (let ((inputs (merge inputs1 inputs2))
113                     (outputs (append outputs 1 outputs2)))
114                   (ptw-loop
115                     (preserves computation
116                       (lambda (inputs) (merge (fn1 inputs) (fn2 inputs)))
117                       inputs outputs))))))))))
118 (define find
119   (reduction-stage computation ;; inlineable after loop fusion
120     (lambda (id list)
121       (deconstruct computation list
122         (cons (cons key value) rest) (key value rest)
123         (if (stage-eq computation key id)
124             value
125             (find id rest))))))
126 (define merge
127   ... like find
```

FIG. 3C

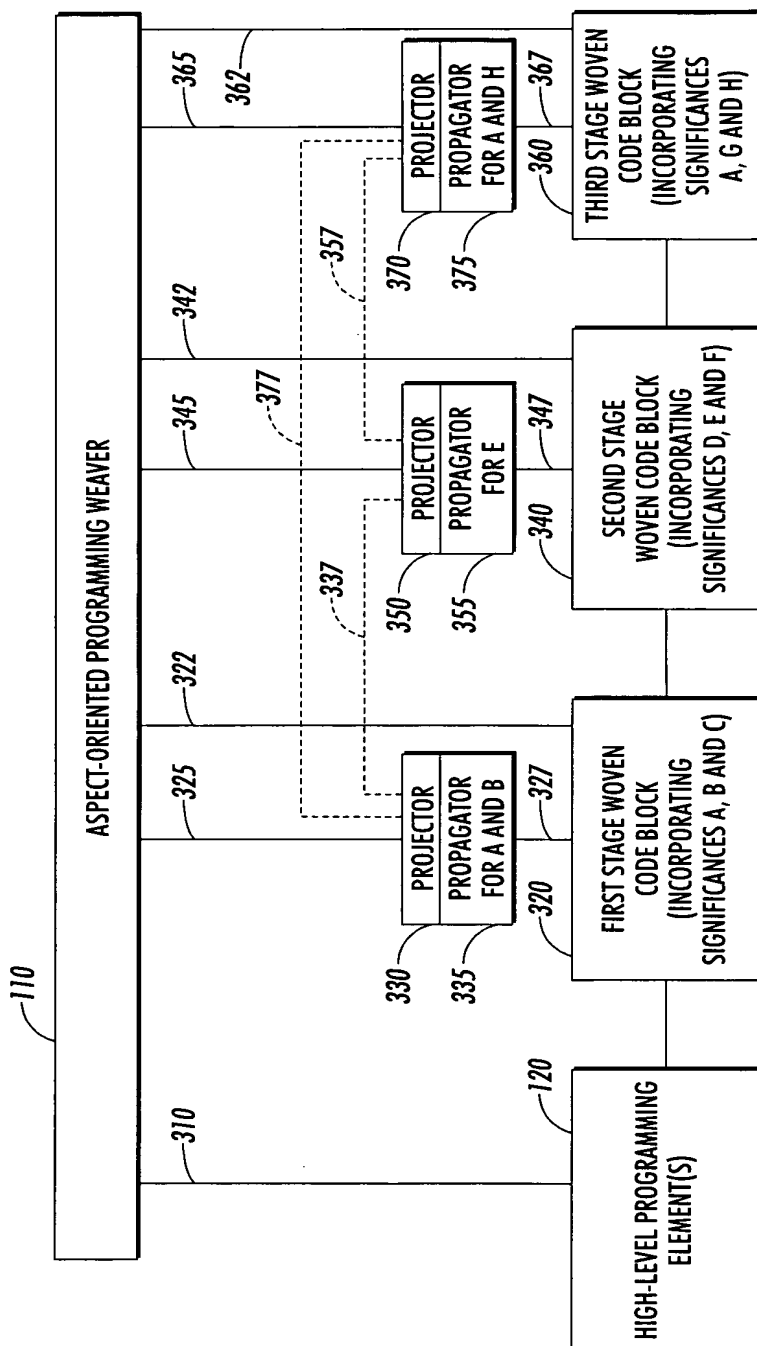


FIG. 4

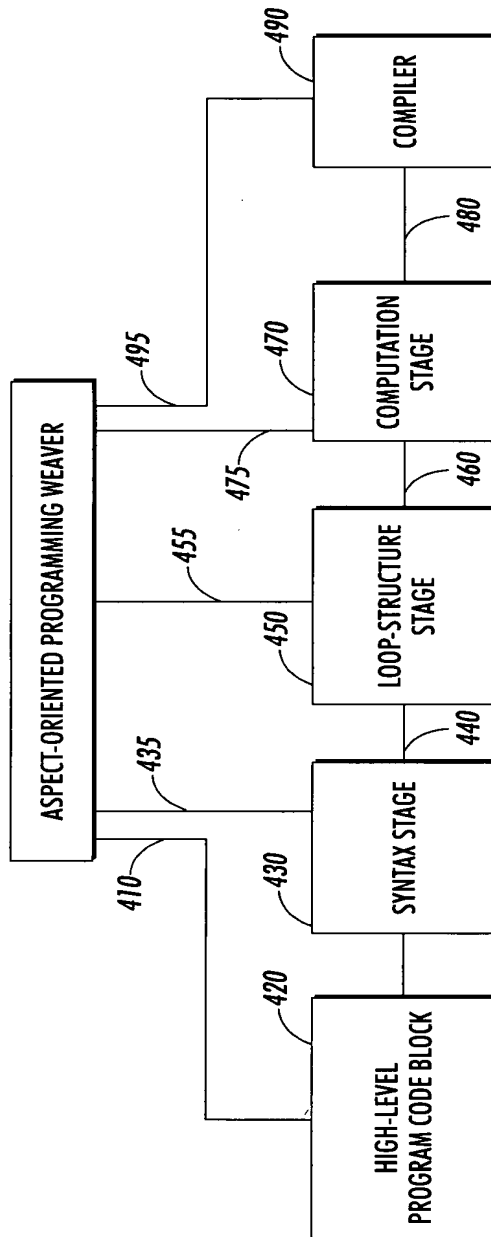


FIG. 5